Conflict Detection in ElectricMake

Eric Melski
Architect
Electric Cloud, Inc.

Abstract

Parallel execution is a popular technique for reducing software build length, and for good reason. These days, multi-core computers have become standard, so there's horsepower to spare, and it's "falling over easy" to implement: just slap a "-j" onto your make command-line, sit back and enjoy the benefits of a build that's 2, 3 or 4 times faster than it used to be. Sounds great!

But then, inevitably, invariably, you run into parallel build problems: incomplete dependencies in your makefiles, tools that don't adequately uniquify their temp file names, and any of a host of other things that introduce race conditions into your parallel build. Sometimes everything works great, and you get a nice, fast, correct build. Other times, your build blows up in spectacular fashion. Then there are the builds that appear to succeed, but in fact generate incorrect outputs, because some command ran too early and used files generated in a previous build instead of the current build.

This is precisely the problem ElectricMake was created to solve — it gives you fast, reliable parallel builds, regardless of how (im)perfect your makefiles and tools are. If the build works serially, it will work with ElectricMake, but faster.

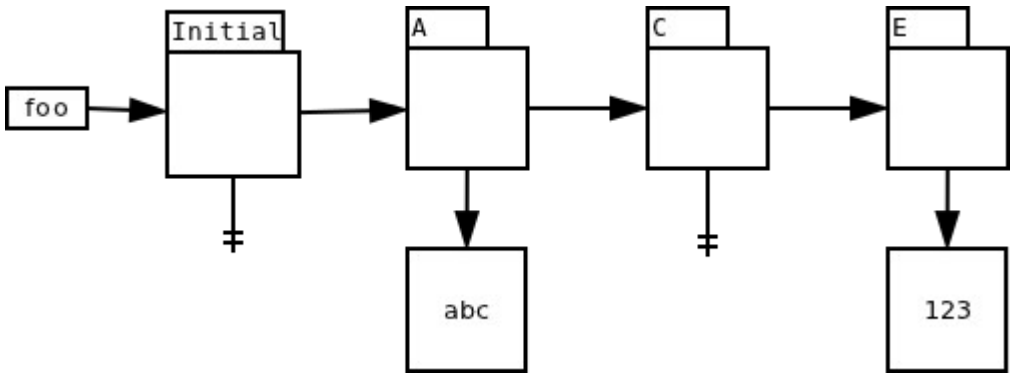
In this talk, I'll explain *how* ElectricMake ensures correct builds, including the ins-and-outs of basic conflict detection, and exceptions to the algorithm that enable higher performance.

Conflict detection

The technology that enables emake to ensure reliable parallel builds is called *conflict detection*. Although there are many nuances to its implementation, the concept is simple. First, track every modification to every file accessed by the build as a distinct version of the file. Then, for each job run during the build, track the files used and verify that the job accessed the same versions it would have had the build run serially. Any mismatch is considered a *conflict*. The offending job is discarded along with any filesystem modifications it made, and the job is rerun to obtain the correct result.

The versioned file system

At the heart of the conflict detection system is a data structure known as the versioned file system, in which emake records every version of every file used over the lifetime of the build. A version is added to the data structure every time a file is modified, whether that be a change to the content of the file, a change in the attributes (like ownership or access permissions), or the deletion of the file. In addition to recording file state, a version records the job which created it. For example, here's what the version chain looks like for a file `foo` which initially does not exist, then is created by job A with contents "abc", deleted by job C, and recreated by job E with contents "123":



Jobs

Jobs are the basic unit of work in emake. A job represents all the commands that must be run in order to build a single makefile target. In addition, every job has a *serial order* — the order in which the job would have run, had the build been run serially. The serial order of a job is dictated by the dependencies and structure of the makefiles that make up the build. Note that for a given build, the serial order is deterministic and unambiguous — even if the dependencies are incomplete, there is exactly one order for the jobs when the build is run serially.

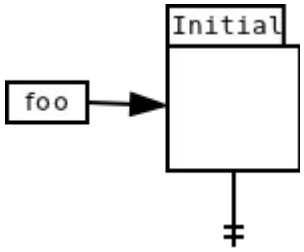
With the serial order for every job in hand, deciding which file version should be used by a given job is simple: just find the version created by the job with the greatest serial order that precedes the job accessing the file. For example, using the version chain above (and assuming that the jobs' names reflect their serial order), job B should use the version created by job A, while job D should see the file as non-existent, thanks to the version created by job C.

A job enters the *completed* state once all of its commands have been executed. At that point, any filesystem updates created by the job are integrated into the versioned filesystem, but, critically, they are not pushed to the real filesystem. This gives emake the ability to discard the updates if the job is later found to have conflicts.

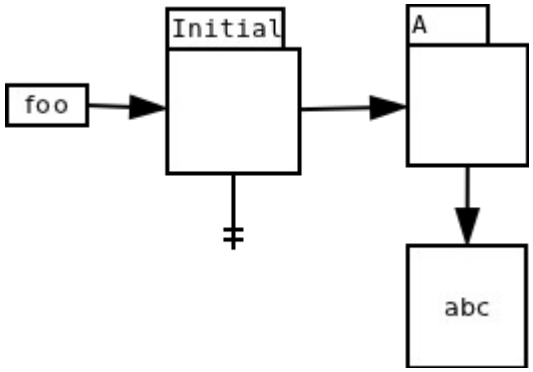
Each job runs against a virtual filesystem called the *Electric File System* (EFS), rather than the real filesystem. The EFS serves several important functions. First, it is the means by which emake tracks file accesses. Second, it enables commands in the build to access file versions that exist in the versioned filesystem, but not yet on the real filesystem. Finally, it isolates simultaneously running jobs from one another, eliminating the possibility of crosstalk between commands.

Detecting conflicts

With all the data emake collects — every version of every file, and the relationship between every job — the actual conflict check is simple: for each file accessed by a job, compare the *actual version* to the *serial version*. The *actual version* is the version that was used when the job ran; the *serial version* is the version that would have been used, if the build had been run serially. For example, consider job B which attempts to access file `foo`. At the time that B runs, the version chain for `foo` looks like this:



Given that state, B will use the initial version of `foo` — there is no other option. The initial version is therefore the actual version used by job B. Later, job A creates a new version of `foo`:



Since job A precedes job B in serial order, the version created by job A is the correct serial version for job B. Therefore, job B has a conflict.

If a job is determined to be free of conflicts, the job is *committed*, meaning any filesystem updates are at last applied to the real filesystem. Any job that has a conflict is *reverted* — all versions created by the job are marked invalid, so subsequent jobs will not use them. The conflict job is then *rerun* in order to generate the correct result. The rerun job is committed immediately upon completion.

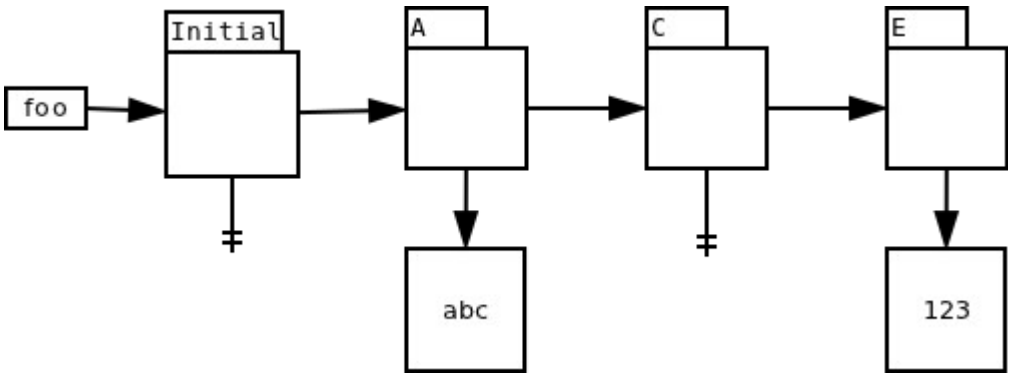
Conflict checks are carried out by a dedicated thread which inspects each job in strict serial order. That guarantees that a job is not checked for conflicts until after every job that precedes it in serial order has been successfully verified free of conflicts — without this guarantee, we can’t be sure that we know the correct serial version for files accessed by the job. Similarly, this ensures that the rerun job, if any, will use the correct serial versions for all files — so the rerun job is sure to be conflict free.

Exceptions to the base algorithm

The basic conflict detection algorithm is simple, which is one of its strengths. But if ElectricMake strictly adhered to the simple definition of a conflict, many builds would be needlessly serialized, sapping performance. Over the years we've made a variety of tweaks to the core algorithm, adding support for special cases to improve performance.

Non-existence conflicts

One obvious enhancement is to ignore conflicts when the two versions are technically different, but effectively the same. The simplest example is when there are two versions of a file which both indicate non-existence, such as the initial version and the version created by job C in this chain for file `foo`:



Suppose that job D, which falls between C and E in serial order, runs before any other jobs finish. At runtime, D sees the initial version, but strictly speaking, if it had run in serial order it would have seen the version created by job C. But the two versions are functionally identical — both indicate that the file does not exist. From the perspective of the commands run in job D, there is no detectable difference in behavior regardless of which of these two versions was used. Therefore emake can safely ignore this conflict.

Directory creation conflicts

A common make idiom is `mkdir -p $(dir $@)` — that is, create the directory that will contain the output file, if it doesn't already exist. This idiom is often used like so:

```
$(OUTDIR)/foo.o: foo.cpp
        @mkdir -p $(dir $@)
        @g++ -o $@ $^
```

Suppose that the directory does not exist when the build starts, and several jobs that employ this idiom start at the same time. At runtime they will each see the same filesystem state — namely, that the output directory does not exist. Each job will therefore create the directory. But in reality, had these jobs run serially, only the first job would have created the directory; the others would have seen the version created by the first job, and done nothing with the directory themselves. According to the simple definition of a conflict, all but the first (serial order) job would be considered in conflict. For builds without a history file expressing the dependency between the later jobs and the first, the performance impact would be disastrous.

Prior to Accelerator 5.4, there were two options for avoiding this performance hit: use a good history file, or arrange for the directories to be created before the build runs. Accelerator 5.4 introduced a refinement to the conflict detection algorithm which enables emake to suppress the conflict between jobs that both attempt to create the same directory, so even builds with no history file will not get conflicts in this scenario, without sacrificing correctness. *(NB: you need not take special action to enjoy the benefits of this improvement).*

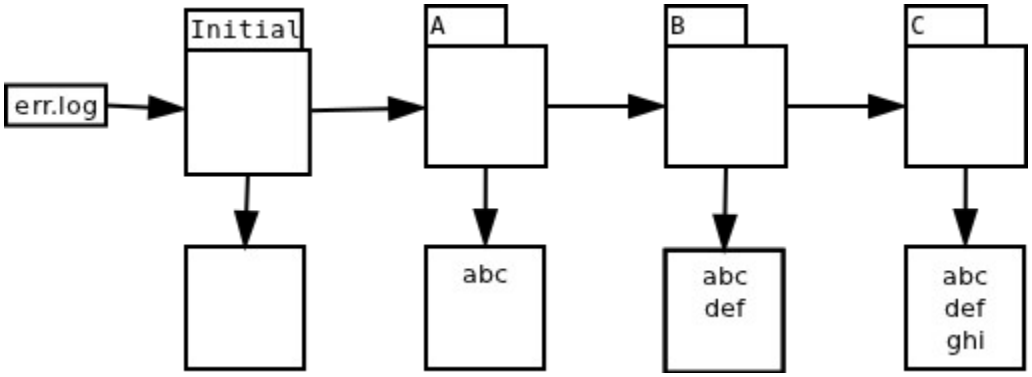
Appending to files

Another common idiom is to append error messages to a log file as the build proceeds:

```
$(OUTDIR)/foo.o: foo.cpp
        @g++ -o $@ $^ 2>> err.log
```

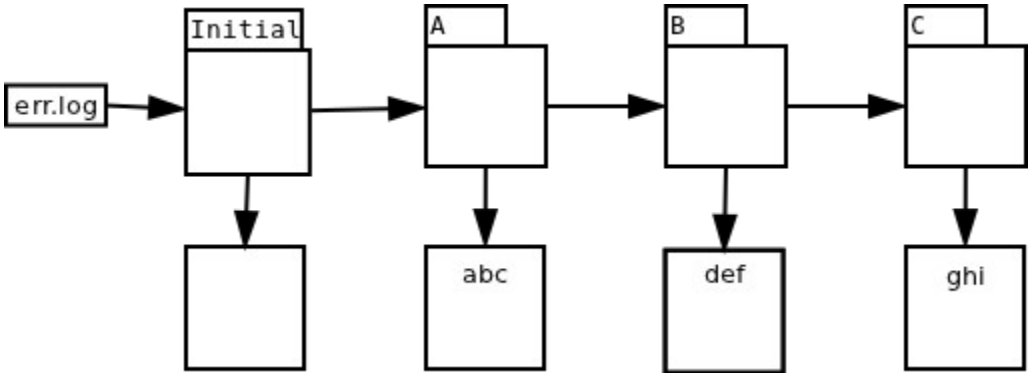
Implicitly, each append operation is dependent on the previous appends to the file — after all, how will you know the file offset of the new content if you don't know how big the file was to

begin with? In terms of file versions, you can imagine a naive implementation treating each append to the file as creating a complete new version of the file:



The problem of course is that you'll get conflicts if you try to run all of these jobs in parallel. Suppose all three jobs, A, B and C start at the same time. They will each see the initial version, an empty file, but if run serially, only A would have seen that version. B would have seen the version created by A; C would have seen the version created by B.

This example is particularly interesting because emake cannot sort this out on its own: as long as the usage reported for `err.log` is the very generic "this file was modified, here's the new content" message normally used for changes to the content of an existing file, emake has no choice but to declare conflicts and serialize these jobs. Fortunately, emake is not limited to that simple usage record. The EFS can detect that each modification is strictly appending to the file, with no regard to the prior contents, and include that detail in the usage report. Thus informed, emake can record *fragments* of the file, rather than the entire file content:



Since emake now knows that the jobs are not dependent on the prior content of the file, it need not declare conflicts between the jobs, even if they run in parallel. As emake commits the modifications from each job, it stitches the fragments together into a single file, with each fragment in the correct order relative to the other pieces.

Directory read conflicts

Directory read operations are interesting from the perspective of conflict detection. Consider: what does it mean to *read* a directory? The directory has no content of its own, not in the way that a file does. Instead, the "content" of a directory is the list of files in that directory. To check for conflicts on a directory read, emake must check whether the list of files that the job actually saw matches the list that it would have seen had it run in serial order — in essence, doing a simple conflict check on each of the files in the directory.

That's conceptually easy to do, but the implications of doing so are significant: it means that emake will declare a conflict on the directory read anytime any other job creates or deletes any file in that directory. Compare that to reads on ordinary files: you only get a conflict if the read happens before a write operation *on the same file*. With directories, you can get a conflict for modifications to *other files entirely*.

This is particularly dangerous because many tools actually perform directory reads under-the-covers, and often those tools are not actually concerned with the complete directory contents. For example, a job that enumerates files matching `*.obj` in a directory is only interested in files ending with `.obj`. The creation of a file named `foo.a` in that directory should not affect the job at all.

Another nasty example comes from utilities that implement their own version of the `getcwd()` system call. If you're going to roll your own version, the algorithm looks something like this:

1. Let `cwd` = ""
2. Let `current` = "."
3. Let `parent` = "./.."
4. stat `current` to get its inode number.
5. read `parent` until an entry matching that inode number is found.
6. add the name from that entry to `cwd`
7. Set `current` = `parent`.
8. Set `parent` = `parent` + "/.."
9. Repeat starting with step 4.

By following this algorithm the program can construct an absolute path for the current working directory. The problem is that the program has a read operation on *every directory* between the current directory and the root of the filesystem. If emake strictly adhered to conflict checking on directory reads, a job that used such a tool would be serialized against *every* job that created or deleted *any* file in *any* of those directories.

For this reason, emake *deliberately* ignores conflicts on directory read operations by default. Most of the time this is safe to do, surprisingly — often tools do not need a completely accurate list of the files in the directory. And in every case I've seen, even if the tool *does* require a perfectly correct list, the tool follows the directory read with reads of the files it finds. That means that you can ensure correct behavior by running the build one time with a single agent, to ensure the directory contents are correct when the job runs. That run will produce history based on the file reads, so subsequent builds can run with many agents and still produce correct results. Starting with Accelerator 6.0, you can also use `--emake-readdir-conflicts=1` to force emake to honor directory read conflicts.

Conclusion

Getting parallel builds that are fast is easy: just add `-j` to your `make` invocation. Getting parallel builds that are both fast *and* reliable is another story altogether. As you've seen, the core conflict detection algorithm in ElectricMake is simple, but after many years and hundreds of thousands of builds, we've enhanced that simple algorithm in a variety of special cases to provide even better performance. Future releases of ElectricAccelerator will include even more refinements to the algorithm.